Strategies for Solving Linear Systems of Equations Using Chapel *

Richard F. Barrett and Stephen W. Poole
Future Technologies Group
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37931

Abstract

Methods for solving linear systems of equations are at the heart of many computational science applications. Examples include science domains such as astrophysics, biology, chemistry, fusion energy, power system networks, and structural engineering, employing a diverse set of modeling approaches, such as computational fluid dynamics, finite element modeling, and linear programming. In this report we discuss how the global view programming language Chapel, being developed as part of the Cray Cascade project, may be used to express algorithms for solving these systems. Our focus is on sparse matrices, and the expression of the matrix-vector product operation required by Krylov subspace solution algorithms. We include an application that generates a dense linear system.

1 Introduction

Scientific application programs often require the solution of linear systems of the form

$$Ax = b, \tag{1}$$

for $A \in \mathbf{R}^{n \times n}$ and x and $b \in \mathbf{R}^n$. The effort spent solving these systems can consume a significant portion of overall program execution time. As computing environments reach peta-scale, with talk already of exa-scale, the issues involved in solving these systems are magnified, and will require the involvement of a variety of experts in order to successfully address them.

Chapel is an emerging programming language[8] designed to present the code developer with a global view of the computations that make up a scientific application program designed for use on a distributed memory parallel processing architecture. In this report we explore the use of Chapel for posing linear equation solution methods within the context of some scientific application areas.

We begin with a brief discussion of methods for solving linear systems and the issues that often dominate the performance of these methods. Next we present an overview of Chapel, with a focus on the syntax and semantics used for this work along with an introduction to alternate programming languages. We then illustrate the use of Chapel in scientific application areas. Finally we offer our conclusions and discuss our future research.

*This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

2 Motivation

The time spent solving a system of linear equations is a function of two things: first is the choice of algorithm, and second is the implementation of that algorithm. For dense linear systems, we can apply "black-box" methods, for example as found in the ScaLAPACK library[17]. Here performance is a function of the quality of the BLAS library[13] and the inter-process communication protocol used by the BLACS library[14]. For sparse linear systems, the choices are more complex: sparse direct or iterative, multi-grid, etc. Sparse direct methods, also captured in libraries (e.g. [28, 1, 37]), will not be discussed in this report, nor will multigrid.

Iterative methods are also available in libraries (e.g. [20, 4, 18, 3]). However, the key to the performance of iterative solvers still usually resides with the code developer and problem set up. In particular, the choice of algorithm and preconditioner determines the rate of convergence, and the implementation of the matrix-vector product and preconditioner determines the computational performance.

Regardless of the solution method, the manner in which the problem is described and translated to a representation for use on a computer plays a significant role in the computation performance of a solution algorithm. (For ScaLAPACK, we include the time and effort spent getting the data into the necessary 2d block cyclic format, often a non-trivial task.)

Because of the various issues involved in solving linear systems of equations at the peta-scale, we want the participation of experts from areas such as numerical analysis, numerical linear algebra, and computer science. Unfortunately, the complexity of the programming mechanisms have dissuaded these sorts of collaborations. From one perspective, this is a confusing situation: the algorithms are relatively easy to understand and apply. For example, a wide variety of Krylov subspace solvers have been developed for computing the approximate solution iteratively, as illustrated by the Conjugate Gradient method[21] shown in Figure 1. From a computational perspective, these

```
Compute $r_0 = b - Ax_0$ for some initial guess $x_0$
for $i = 1, 2, \ldots$
    solve $Mz_{i-1} = r_{i-1}$          (Preconditioning)
    $\rho_{i-1} = r_{i-1}^T z_{i-1}$
    if $i = 1$
        $p_1 = z_0$
    else
        $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
        $p_i = z_{i-1} + \beta_{i-1} p_{i-1}$
    endif
    $q_i = Ap_i$                        (Matrix-vector product)
    $\alpha_i = \rho_{i-1}/p_i^T q_i$
    $x_i = x_{i-1} + \alpha_i p_i$
    $r_i = r_{i-1} - \alpha_i q_i$
    check convergence; continue if necessary
end
```

Figure 1: Conjugate Gradient method
If $A \in \mathbf{R}^{n \times n}$ is symmetric positive definite and $b \in \mathbf{R}^n$, then the following algorithm, using symmetric positive definite preconditioner M, generates approximations x_i to $x \in \mathbf{R}^n$ of $Ax = b$. Computations shown in blue represent the direct interaction of the algorithm with the linear equations.

algorithms add and scale vectors (perfectly parallel) and compute inner products and vector norms (global communication). It is through the matrix-vector product (and optionally preconditioning) that these algorithms interact with the physical problem, which often presents significant challenges.

3 Overview of Chapel

Scientific application programs designed for use on parallel processing architectures are typically written using Fortran, C, or C++, with the interaction of the distributed memory, distinct name spaces managed using functionality defined by the MPI specification[36, 19]. This approach requires the code developer to explicitly manage the distribution and movement of data among the parallel processes, each of which has its own distinct address space.

Partitioned Global Address Space (PGAS) languages have been developed in order to address some of the difficulties perceived with this approach. For example, Co-Array Fortran (CAF)[30] makes data globally accessible via co-array load and store semantics, though it still places the responsibility for distributing and moving the data between the parallel processes on the code developer. Unified Parallel C (UPC) [16] extends the C programming language to include a shared address space view of computation, but it inherits C's limited support for arrays, with no real support for multidimensional arrays. Titanium[22] extends Java$^{\mathrm{TM}}$to include an explicit parallel Single-Program-Multiple-Data (SPMD) model.

Each of these programming models provide a "fragmented view" of parallel computation in that they require the code developer to explicitly manage the interaction of the parallel processes as well as the overall data layout. Chapel provides a "global view" of the computation and associated data. The goal is to provide an easier means for writing code for execution on a parallel processing architecture.

We've seen this approach before, and the results, especially with regard to performance, have not been satisfying for a majority of the scientific computing community. For example, OpenMP[10] presents such a view through the definition of a set of compiler directives and associated syntax, but is limited to regions of physically shared memory in a node. It may be combined with MPI to link multiple distributed shared memory regions such as on a cluster of SMP nodes, which then introduces the fragmented view. High Performance Fortran (HPF)[32] provides a global view, but is constrained to the SPMD model and supports only a single data structure (the Fortran array). This rules out a broad range of problems defined on irregular domains as well as the implementations described in Section 4.2.

Chapel is attempting to combine the strengths of successful programming languages while avoiding their weaknesses. Most fundamentally, Chapel provides a means for defining global data structures. These *domains* are constructs that provides the code developer with a means for configuring data structures that enable a more natural mapping of computations to the parallel processes, including distribution of data and associated inter-process data sharing. The overall goal is to combine a global view of the program with the tools necessary for injecting high-level programmer "intent" that the compiler cannot easily discover in more traditional programming models.

At the time of this writing, the Chapel language specification[9] is at version 0.702. A prototype compiler (pre-release version 0.4) has been provided to a small group of programmers who are gaining experience and providing feedback to the Chapel developers. An updated release is expected in summer 2007.

Two other notable global view language development efforts are also underway. Fortress[2] endeavors to present a mathematically based syntax to the code developer. X10[23] extends Java$^{\mathrm{TM}}$. We are investigating these languages in a manner similar to that described in this report.

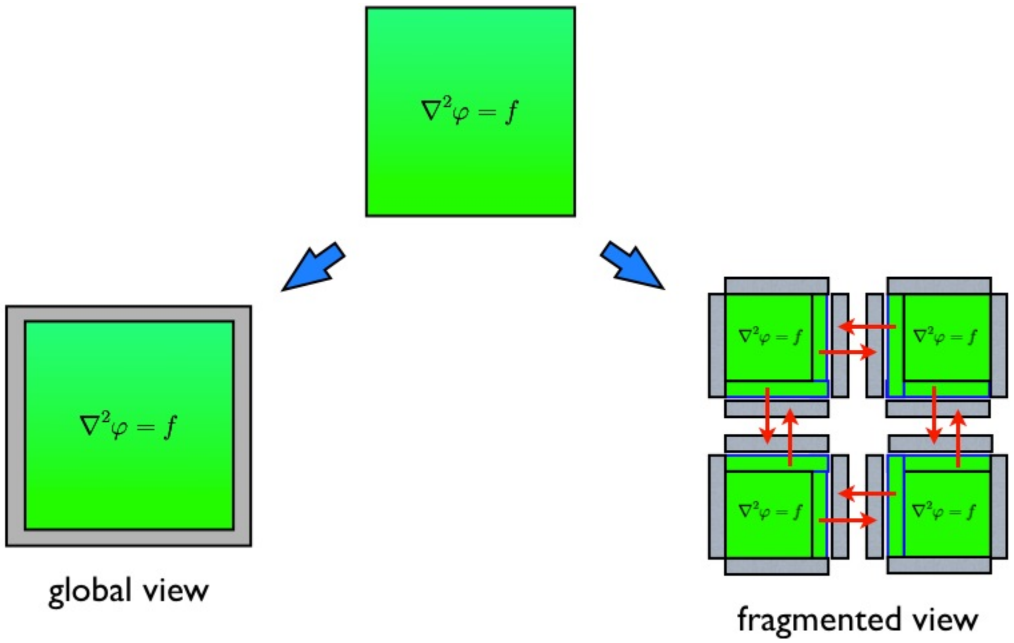


Figure 2: Global view vs. fragmented view parallel programing model.
Consider a partial differential equation (here Poisson's Equation) defined on a two dimensional domain, illustrated by the center picture. The fragmented view configuration for applying a solution algorithm on a parallel processing computer is shown on the right. Here the code developer must manage the interaction of the parallel processes as well as the overall data layout, including explicit control over the sharing of data among the individual blocks. This is usually accomplished by surrounding each block with a "halo" in order to control data movement (as indicated by the arrows) and maintain coherency. A global view language such as Chapel captures data associated with the problem in a single structure which it (as well as a fragmented model) may then surround with space for the physical boundary conditions. Although the language may provide a means for conveying information regarding parallelism in the problem (Chapel does[12]), the code developer is not responsible for distributing and sharing data amongst the parallel processes.

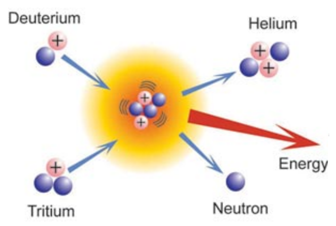
4 Applications

In scientific application programs, linear equations typically originate from the discretized problem space, which may be described as a some regular structure (e.g. rectangular with equally spaced grid points), or an irregular structure described as a graph. The resulting matrix description may stay within the context of this description ("in place"), may be translated into some artificial storage format (e.g. some "compressed" format[4], or 2d block cyclic[17]), or may be approximated (matrix-free methods[26]).

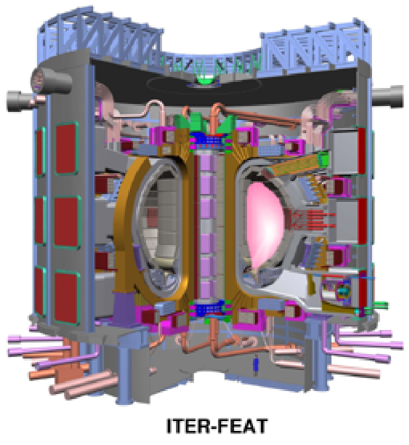
We illustrate a variety of matrix structures in order to discuss various strategies for defining data structures for storing them and the matrix-vector products that operate upon them. We begin with a dense linear system that arises from a fusion energy model. Next, a sparse linear system from a well-known ocean model whose regularity we can exploit. Then we show a broad range of matrix structures that arise from a variety of science and engineering areas, and sketch strategies for defining them using Chapel domains.

4.1 Fusion energy: a dense linear system from a sparse Chapel domain

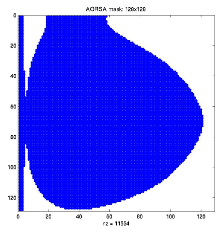
Fusion energy research is a major area of interest for the US Department of Energy. Devices, such as the Tokamak Fusion Test Reactor (TFTR)[27] and the International Thermonuclear Experimental Reactor (ITER)[34] (shown in Figure 3(b)) provide important experimental platforms for studying



(a) Fusion reaction.



(b) ITER fusion energy device.



(c) AORSA Fourier space, with the physical (cavity) space shown in blue.

Figure 3: Fusion energy modeling

the necessary plasma dynamics. The AORSA computer program provides a computational means for studying the behavior of electromagnetic waves, a fundamental factor in these dynamics.

The two- and three-dimensional all-orders spectral algorithms (AORSA) code[24] is a full-wave model for radio frequency heating of plasmas in fusion energy devices. AORSA operates on a spatial mesh, with the resulting set of linear equations solved for the Fourier coefficients. A Fast Fourier Transform algorithm converts the problem to a frequency space (see Figure 3(c)), resulting in a dense, complex-valued linear system, which is solved using functionality provided by the ScaLAPACK[17] or a locally modified version of HPL[33, 11] libraries.

AORSA's Fourier space can be defined using a Chapel arithmetic domain. The physical space is constructed by scanning the Fourier domain and determining inclusion or exclusion of a grid point in the device cavity. These points are defined using a Chapel sparse domain subset of the Fourier domain. A code segment illustrating this idea is shown in Figure 4.

The solution of the dense linear system using HPL in AORSA has achieved 75.1 TFLOPS executing on 11,250 dual core processors of Jaguar, the Cray XT4[25] located at Oak Ridge National Laboratory (ORNL). This scale has allowed researchers to conduct experiments at resolutions previously unattainable. For example, preliminary calculations using 4,096 processors allowed the first simulations of mode conversion in ITER. Therefore there is no need to implement this algorithm using Chapel. Instead, this highlights the importance of interoperability of Chapel with libraries written in other languages and programming models, in this case Fortran with MPI[1].

[1]More fundamentally, this highlights the need for research into the interaction of global view and fragmented view languages.

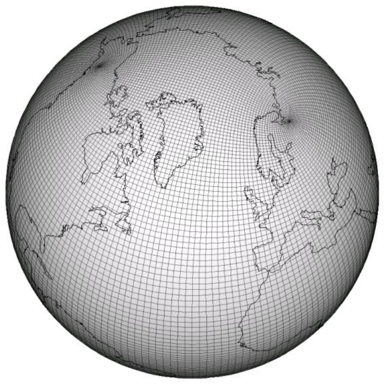
```
const FourierSpace = [ 1.. nnodex, 1.. nnodey ];

var
  fgrid,
  mask
      : [ FourierSpace ] real;

// Work in Fourier space, with grid points in the physical space flagged in array mask, used to define domain PhysSpace:

PhysSpace : sparse subdomain ( FourierSpace ) = [ i in FourierSpace ]

     if ( mask(i) ) then i;

var pgrid :[ PhysSpace ] real;
```

Figure 4: AORSA: Fourier and Physical space defined using Chapel domains

4.2 Ocean modeling: a regular sparse matrix

POP (Parallel Ocean Program) is an ocean circulation model[31]. Developed at Los Alamos National Laboratory, it also serves as the ocean component of the Community Climate System Model (CCSM[6]).

POP models ocean circulation by solving time dependent equations describing fluid motion in three dimensions. Computation takes place on an orthogonal curvilinear coordinate system on a dipole[35] or a tripole grid[29]. The latter is shown in Figure 5.


(a) Tripole grid

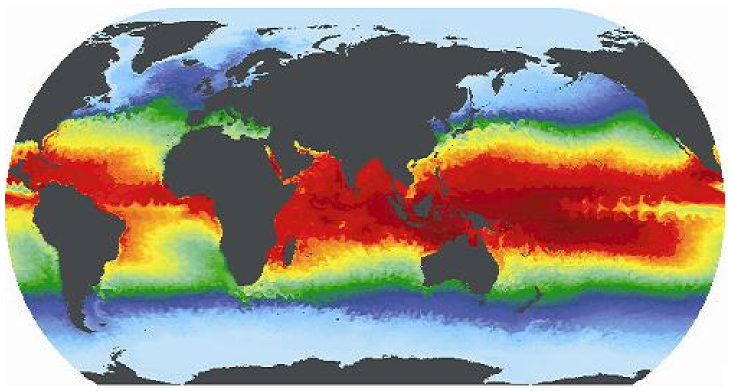

(b) Output

Figure 5: POP ocean discretization using a tripole grid. (These figures are from the POP web pages[31].

Computation of surface pressure requires the solution of the 2-dimensional barotropic equations. The implicit solution method employed in the barotropic phase configures an elliptic equation of the form

$$AF = B \tag{2}$$

where F is a field (in this case surface pressure) and A is the operator defined as

$$AF = a\nabla \cdot (H\nabla F) \tag{3}$$

where a is the cell area. For each model time step, a linear system of equations is generated of the form shown in (1). The solution of this symmetric positive definite sparse system is computed iteratively most often using the Conjugate Gradients (PCG) algorithm, previously shown in Figure 1.

A linear equation in the system is defined as a function of a grid point and the eight grid points immediately adjacent to it. Thus the matrix-vector product, computed as part of each iteration, may be formulated as a nine-point stencil sweep across the grid. The Fortran implementation of this computation is shown in Figure 6. Each processor owns arrays of size `imt` × `jmt` for storing

```
   real (kind=dbl_kind), dimension(imt,jmt), intent(in) ::
 &   X, XOUT,                 ! array to be shifted
 &   CC,CN,CS,CE,CW,          ! weights in each of the nine directions
 &   CNE,CSE,CNW,CSW

   do j = jphys_b, jphys_e
     do i = iphys_b, iphys_e
       XOUT(i,j) = CNE(i,j)*X(i+1,j+1) + CNW(i,j)*X(i-1,j+1) +
 &                 CSE(i,j)*X(i+1,j-1) + CSW(i,j)*X(i-1,j-1) +
 &                 CN (i,j)*X(i  ,j+1) + CS (i,j)*X(i  ,j-1) +
 &                 CE (i,j)*X(i+1,j  ) + CW (i,j)*X(i-1,j  ) +
 &                 CC (i,j)*X(i,j)
     end do
   end do
```

Figure 6: POP's matrix-vector product posed as 2d 9-point stencil computation.

the matrix coefficients and grid points as well as for ghost cells (also called halos) filled using inter-process communication. The subdomain of grid points actually owned by a parallel process is a subset of that space, running from `iphys_b` to `iphys_e` to `jphys_b` to `jphys_e`. The coefficients for each (i, j) linear equation in the system are stored in separate arrays, e.g. `CNE(i,j)`, `CNW(i,j)`, which serve as weights for the stencil.

The MPI implementation requires that the code developer explicitly distribute and move the data between the parallel processes (as previously shown in Figure 2).

Using Chapel we can create, in essense, a translation from Fortran, shown in Figure 7. Note that

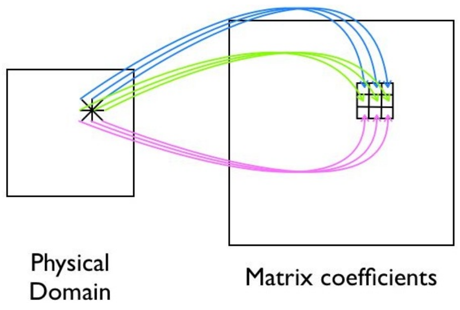
```
const
  PhysicalSpace = [ 1..imt_global, 1..jmt_global ],  // Grid points in the 2d physical domain.
  AllSpace = PhysicalDomain.expand(1);               // Physical domain plus boundary.

var
  X, XOUT,
  CNW, CN, CNE, CW, CC, CE, CSW, CS, CSE    // Weights in each of the nine directions.
     : [ AllSpace ]  real;

// Define neighbors:

const
  NW = (-1,-1), N = (-1,0), NE = (-1,1), W = (0,-1), E = (0,1), SW = (1,-1), S = (1, 0), SE = (1,1);

forall i in PhysicalSpace  do

  XOUT(i) = ( CNW(i)*X(i+NW) + CN(i)*X(i+N) + CNE(i)*X(i+NE) +
              CW(i)*X(i+W ) + CC(i)*X(i)   +  CE(i)*X(i+E ) +
              CSW(i)*X(i+SW) + CS(i)*X(i+S) + CSE(i)*X(i+SE)   );
```

Figure 7: POP's matrix-vector product in Chapel using parameterized tuple arithmetic.

the spaces containing the grid points and coefficients are defined using the global number of grid points `imt_global` and `jmt_global`, with no need to be concerned with ghost space or other issues associated with the parallel processing environment.

An arithmetic domain describes the grid points in the physical domain; from it, we derive a second domain, which describes the grid points as well as space for applying the physical boundary conditions. Arrays are allocated using the latter space, while iteration is controlled by the physical space.

An alternative implementation can be constructed by describing the matrix-vector product stencil operation as a weighted reduction operation. This approach requires that we first reconfigure the storage of the matrix coefficients. We generate and store the matrix elements as sets of 3×3 blocks, where each block represents the matrix coefficients for a physical grid point, as illustrated in Figure 8. That is, the nine arrays storing the matrix coefficients can be replaced with a single "array of



```
const
  PhysicalSpace = [ 1..imt_global, 1..jmt_global ],    // Grid points in the 2d physical domain.
  Stencil = [ -1..1, -1..1 ];                          // 2d 9-pt stencil domain.

var
  Coeff: [ PhysicalSpace ][ Stencil ]  real;            // "Array of arrays" for storing matrix coefficients.
```

Figure 8: The matrix coefficients associated with each grid point are stored as 3×3 blocks in the "array of arrays" defined domain `Coeff`.

arrays" also shown in Figure 8. Coefficients for the i^{th} linear equation are `Coeff(i)(k)` for each `k` in domain `Stencil`.

The sparse matrix-vector product may now be posed as a stencil computation, with matrix coefficients serving as weighting factors, as shown in Figure 9. The operation is addition, signified

```
// Perform matrix-vector product:

forall i in PhysicalSpace do
  XOUT(i) = + reduce [ k in Stencil ] Coeff(i)(k)*X(i+k);
```

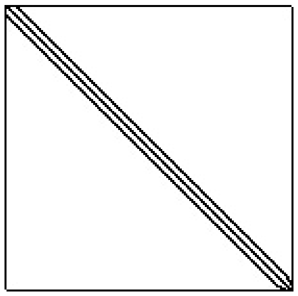
Figure 9: POP's matrix-vector product in Chapel using the reduction operator.

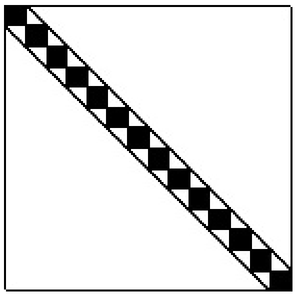
by "+". The scope of the reduction operator is controlled by `[k in Stencil]`, which is shorthand syntax for an expression-level `forall` loop.

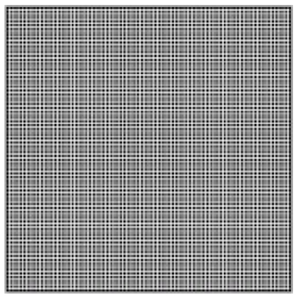
4.3 More examples: structured and unstructured matrices

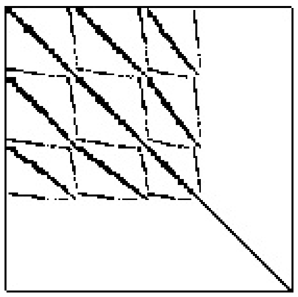
Figure 10 show a variety of matrix structures from a range of science and engineering areas[2]. Some of these systems contain substructures that may be exploited, while other systems show no regular structure. Chapel domains can be constructed to capture these structures, perhaps as a set of substructures.

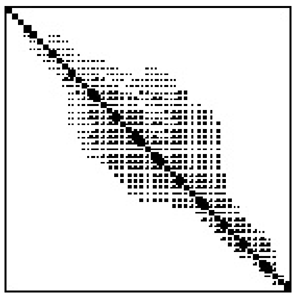
[2]These matrices are found in the Harwell-Boeing set[15], may be downloaded from the Matrix Market web site[7].

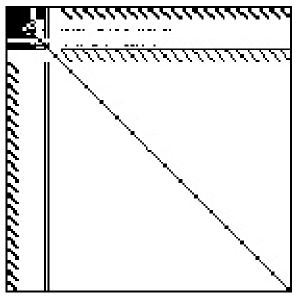

(a) 2D fluid flow in a driven cavity; non-symmetric and indefinite (e30r1000)

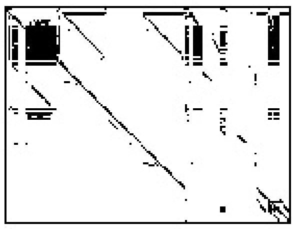

(b) Model of H2+ in an Electromagnetic Field. (qc2534)

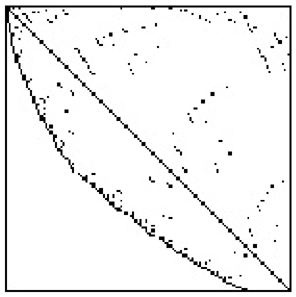

(c) Structural Engineering Matrix (eigenvalue matrix) (bcsstk02)


(d) Air-traffic Control Model: Hessian of the objective function. (zenios)


(e) Astrophysics: Nonlinear radiative transfer and statistical equilibrium (mcca)


(f) Jacobian of a nonlinear system of ODEs modeling a laser. (arc130)


(g) One of fifteen regions in world economic model (wm1)


(h) Simulation of computer systems. (gre115)

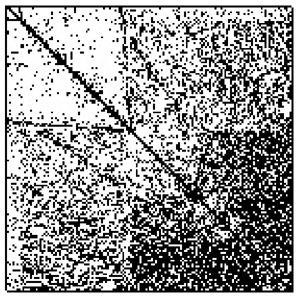

(i) Western US power network. (bcspwr10)

Figure 10: Matrix structures
These graphs represent the structure of matrices from a variety of scientific and engineering areas. The matrices are from the Harwell-Boeing test set, available from the MatrixMarket[7].

Matrix 10(a) is not quite regular (nor symmetric) and thus the matrix-vector product cannot be posed as a stencil. It is ill-conditioned ($O(10^{10})$), so some effort at ensuring strong performance would be desirable.

Matrix 10(b) arises from a physical problem set up to determine the laser-induced molecular resonance states of H_2^+ in an electro-magnetic field. The resulting matrix is complex value, symmetric indefinite. It has a clear block structure (as well as an upper and lower band) that should be exploited. This structure could also form the basis of a preconditioning strategy. We could define a domain that captures the coefficients of a block and the diagonal elements of the rows

Matrix 10(c) is from structure engineering problem, and is actually an eigenvalue matrix. In other common programming languages, such as C or Fortran, this would most likely be represented as a dense matrix in a 2d array. With Chapel, we could define a sparse domain, saving some memory, and still perhaps apply a sparse algorithm effectively.

The other matrices (10(d)-10(i)) have far less to no regularity that could be exploited. However, it would be worth exploring strategies involving subsets of the matrices (notably 10(f)). Otherwise, opaque domains would be necessary in order to describe the matrices as graphs, which allows special algorithms to be applied.

5 Conclusions and Future Work

We have demonstrated the syntax and semantics of Chapel using a variety of scientific applications. Although our focus is on the solution of linear systems, the language capabilities illustrated have broad applicability and appeal within the scientific computing community. In particular, the ability to create customized data structures, and apply operations on them in a manner that is not dependent upon that structure (i.e. polymorphism) is a powerful capability both in terms of programmability and performance potential.

The alert reader will notice that the code that applies the matrix-vector multiplication computation is indepedent of the data structure (domain) definitions. For example, referring back to Figure 9, note that we could change the domain definition of the stencil and the data arrays, yet the computation would remain unchanged. Non-rectangular stencils may be defined using a sparse domain as illustrated in Section 4.1. Both of these issues are discussed in more detail in [5].

We look forward to tracking the progress of the Chapel specification and prototype compiler, both as a means of investigating the expressiveness of the language within the context of important applications and also the performance capabilities and potential.

We are actively investigating the use of Chapel, as well as Fortress and X10, for other classes of computations. More interesting (and more challenging!) are our investigations into how Chapel constructs might influence the development and choice of algorithms in posing computational science experiments.

Acknowledgements

We are grateful to Brad Chamberlain, the Technical Lead of the Chapel project, for invaluable discussions, advice, and code review. He has taken seriously our many discussions regarding our perceptions of the value of existing as well as potential functionality in the Chapel language. We extend this gratitude to all members of the Chapel team who, through Brad, aided our efforts. We look forward to continued interactions.

References

[1] Multifrontal Massively Parallel Solver (MUMPS version 4.7) user's guide. Technical report, April 2007. `http://graal.ens-lyon.fr/MUMPS/userguide_4.7.pdf`.

[2] E. Allen, D. Chase, J. Hallet, V. Luchangco, J. Maessen, S. Ryu, G. L. Steele Jr, and S. Tobin-Hochstadt. The Fortress language specification, version 1.0.β. Technical report, Sun Microsystems, Inc., 2007.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[4] R.F. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J.J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[5] R.F. Barrett, P.C. Roth, and S.W. Poole. Finite Difference Stencils Implementing Using Chapel. 2007. Submitted.

[6] M.B. Blackmon, B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, J. Hurrell, and et al. The Community Climate System Model. *BAMS*, 82, 2001.

[7] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard Barrett, and Jack J. Dongarra. The matrix market: A web resource for test matrix collections. In *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman and Hall. `http://math.nist.gov/MatrixMarket`.

[8] B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel programming and the Chapel language. *International Journal on High Performance Computer Applications*, To appear, 2007.

[9] Cray, Inc. Chapel language specification 0.702. `http://chapel.cs.washington.edu`, 2006.

[10] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[11] Ed F. D'Azevedo, 2007. Private communication.

[12] R.E. Diaconescu and H.P. Zima. An approach to data distribution in Chapel. *International Journal on High Performance Computer Applications*, To appear, 2007.

[13] J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 16:1–17, 1990.

[14] J.J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Lapack working note: A users' guide to the blacs. Technical report, Computer Science Department, University of Tennessee, 1993.

[15] I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[16] T.A. El-Ghazawi, W.W. Carlson, and J.M. Draper. UPC language specification, version 1.1.1. `http://www.gwu.edu/~upc/documentation.html`.

[17] Blackford et al. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[18] R.D. Falgout, J.E. Jones, and U.M. Yang. The Design and Implementation of hypre, a library of Parallel High Performance Preconditioners. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, 51, pages 267–294. Springer-Verlag, 2006.

[19] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference: Volume 2 - The MPI-2 Extentions.* The MIT Press, 1998.

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R.J. Hoekstra, J. J. Hu, T.G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R. S. Tuminaro, J.M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Transactions on Mathematical Software*, 31:397–423, September 2005.

[21] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[22] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, University of Califoria, Berkeley, 2005.

[23] IBM. Report on the experimental language X10, draft v 0.41. IBM TJ Watson Research Center, `http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html`, February, 2006.

[24] E.F. Jaeger, L.A. Berry, E. D'Azevedo, D. B. Batchelor, and M. D. Carter. All-orders spectral calculation of radio-frequency heating in two-dimensional toroidal plasmas. *Phys. Plasmas*, 8, 2001.

[25] Jaguar. Cray XT4 at Oak Ridge National Laboratory. `http://info.nccs.gov/resources/jaguar`.

[26] D. A. Knoll and D. E. Keyes. Jacobian-free newton-krylov methods: A survey of approaches and applications. *J. Comp. Phys.*, 193:357–397, 2004.

[27] Princeton Plasma Physics Laboratory. Tokamak fusion test reactor. `http://www.pppl.gov/projects/pages/tftr.html`.

[28] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[29] R.J. Murray. Explicit generation of orthogonal grids for ocean models. *Journal of Computational Physics*, 251, 1996.

[30] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998. `http://www.co-array.org`.

[31] Parallel Ocean Program. (POP). `http://climate.lanl.gov/Models/POP`.

[32] High Performance Fortran Forum. High Performance Fortran language specification. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

[33] A. Petitet, R.C. Whaley, J.J. Dongarra, and A. Cleary. HPL: A portable high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/index.html`, January 2004.

[34] The ITER project. `http://www.iter.org`.

[35] R.D. Smith and S. Kortas. Curvilinear coordinates for global ocean models. Technical Report LA-UR 95-1146, Los Alamos National Laboratory, 1995.

[36] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition.* The MIT Press, 1998.

[37] Boeing Phantom Works. Spooles 2.2 : Sparse object oriented linear equations solver. `http://www.netlib.org/linalg/spooles/spooles.2.2.html`.